Aalto University
School of Science
Master's Programme in ICT Innovation

Bernát Kalló

Extending the *Cascading Style Sheets (CSS)* Language with Programming Constructs

Master's Thesis

Espoo, Jun 16, 2015

Supervisors: Professor Heikki Saikkonen, Aalto University School of Science
Kitlei Róbert, ELTE Faculty of Informatics

Instructor: Ádám Siklósi, 4D Soft kft.

Aalto University
School of Science

Aalto University School of Science Master's Programme in ICT Innovation	ABSTRACT OF THE MASTER'S THESIS

Author: Bernát Kalló

Title: Extending the Cascading Style Sheets (CSS) Language with Programming Constructs

Number of pages: 73	Date: 16.6.2015	Language: English

Professorship: Software Systems	Code: T-106

Supervisor: Professor Heikki Saikkonen and Kitlei Róbert

Instructor: Ádám Siklósi

Abstract:

I designed the FunCSS programming language, which is a Turing-complete extension of the Cascading Style Sheets language. It lets web developers define custom functions using embedded JavaScript code fragments. Contrary to other CSS extensions, FunCSS is compiled to JavaScript and executed in the web browser. FunCSS can simplify the implementation of modern web designs which contain interactive animations.

I designed the syntax of FunCSS based on a survey that I conducted among web developers, to optimize its syntax for developer comfort. I designed FunCSS with the long-term goal to turn it into a platform for web browser compatibility libraries. FunCSS has become an interesting language by itself. It combines the rule-based and the functional reactive programming paradigms. The type system of FunCSS is based on regular grammars, and its elemental types include untagged union types, physical units of measure and percentages.

Keywords: web design, CSS, language design, functional reactive programming

Table of Contents

Preface

This thesis is part of my final degree project at EIT ICT Labs Master School. My academic major was Service Design and Engineering. At Aalto university, Finland and ELTE, Hungary, I was fortunate to study the principles and practices of on how to design services in general, beyond the field of information technology, and also the theory and practices of building distributed computer systems.

Although my research project was not about a service, I sought to apply the principles of both design and engineering in it. I sought to create a programming language, with an intuitive and human-centered mindset of a designer, and with the foresight and precision of an engineer. I combined all this with the never ending curiosity of a scientist.

I used science, design and engineering in this thesis, too. My goal was to fill it with high scientific value, but at the same time make it well-designed both aesthetically and practically. Producing the desired output involved some engineering as well. I hope that you, honourable reader, will enjoy reading this thesis at least as much as I enjoyed writing it, and that you will find it beneficial for you.

Glossary

at-keyword
: A token in CSS that starts with `@`, e.g. `@import`, `@font-family`. It starts an at-rule.

at-rule
: A special rule in CSS, usually used to give special instructions to the web browser. It starts with an at-keyword.

CSS preprocessor
: A program that generates CSS from an input file, most often from an extended version of CSS.

component value
: A basic type value, a block or a functional notation [26, s5.4.6].

delimiter
: One of the `! # $ % & * + - . / < = > ? @ ^ ` | ~` CSS tokens.

delimiter-like token
: One of the `, : ; ~= |= ^= $= *= || <!-- -->` CSS tokens. Each of these have their own category in [26].

functional notation
: A structured value in CSS with a name and an argument, e.g. `rotate(90deg)`. If a **function** is declared with the same name and an appropriate return type, the functional notation is a function call (e.g. the functional notation `calc(1 + 2)` is a function call that returns the number `3`).

ident
: Identifier in CSS. It can include letters, digits, underscores and hyphens, and should not start with a number.

keyword
: An ident with a special meaning in a property value. CSS has hundreds of keywords, like `auto`, `normal`, `left` etc.

qualified rule
: A CSS rule that is not an at-rule. Qualified rules are the "normal" CSS rules.

Section 1
Introduction

The Cascading Style Sheets (CSS) language was invented in 1994 by Håkon Wium Lie, for defining the style sheets of web documents [1]. Its first specification [17] stated this:

> *We do not expect CSS to evolve into:*
>
> - *a programming language.*

In this thesis, however, I present my research project for designing and implementing a full-featured programming language, which is a superset of CSS. The *FunCSS* language that I have designed, besides all the features of CSS, supports the definition of custom functions. Along with the language, I designed a standard library that contains several utility functions.

I designed the FunCSS language with the goal that it should be a useful tool in commercial web development. It should help web developers to effectively code interactive animations, and to solve browser compatibility problems. Also, FunCSS can be of interest to academia, as it is based on a combination of two rarely used programming paradigms, and its type system also includes some rather unusual features. How these features can be used effectively can be a subject of further research.

In the following part of the introduction, I give some examples of how FunCSS can be used (1.1). Then I give brief overview of how the research has formed (1.2). Finally, I summarise the achievements of the thesis (1.3).

After the introduction, in Section 2, I summarise related work done in the topic. In Section 3, I present an analysis of the CSS language. In Section 4, I describe how I designed the FunCSS language. In Section 5, I evaluate my work and present possible future research.

1.1 Motivational examples

1.1.1 Interactive animation

As a simple example, let us create a text that always follows the mouse pointer. This can be achieved by using the `page-mouse-x()` and `page-mouse-y()` functions from the standard library of FunCSS. The following FunCSS code and HTML code will create the effect that the text follows the mouse pointer, as illustrated in the picture on the right.

Code 1.1 FunCSS example: Text that follows mouse

```
.mouseFollower {
  position: absolute;
  left: page-mouse-x();
  top: page-mouse-y();
}
```


```
<span class="mouseFollower">This text follows the mouse</span>
```

1.1.2 Timed animation

Let us create a digital clock that shows the current time in a *hour:min:sec* format. We can use the `hours()`, `minutes()`, `seconds()` standard library functions, which return the hours, minutes and seconds of the current time as integers. Note the explicit conversions to strings in the code below—FunCSS does not do implicit type conversions.

Code 1.2 FunCSS example: a digital clock

```
.clock::before {
  content: string(hours()) ":"
           string(minutes()) ":"
           string(seconds());
}
```

21:24:55

```
<span class="clock"></span>
```

1.2 The research idea

During the past 3.5 years while I worked in web development, I often faced web styling tasks that I could not solve with CSS. A part of this was due to my ignorance about CSS features, and another part was due to the lack of expressive power of CSS.

My interest in using CSS for sophisticated styling started to rise in 2012. In early 2012 I read a blog post [2] of a web developer Nicole Sullivan, about the software technology of CSS. She starts her post as follows:

> *The cascade is something like a new data structure, and the ways for dealing with it are algorithms you never learned in school.*

As I understood, she argues that many software engineers have difficulties with CSS because they are not familiar with the unique programming paradigm of CSS.

The thoughts in this blog post made me start to reflect about CSS as a (fairly unusual) programming language. I wanted to understand what programming paradigm it follows and how I could use this paradigm effectively in web development. As CSS is not a full-featured programming language, I posed the following question:

What would CSS be like if it was a full-featured programming language? How could it be better used for web styling?

I answered this question as follows:

Most full-featured, general-purpose programming languages contain support for creating abstractions: variables, subroutines, classes, modules etc. Similarly CSS, if extended into a full-featured programming language, should also contain support for creating abstractions, to ease everyday tasks in web development.

Many developers have invented technologies for adding abstractions to CSS. I will present some of these approaches in 2.1.

CSS already contains abstractions like functions, selectors (see 3.3), shorthand properties [19, s1.4.3], named colours [19, s4.3.6], pseudoelements [19, s5.10], and media queries [22]. However, all of these can only be built-ins, i.e. only the specifications can define new abstractions of these kinds. However, it might

be useful if the author himself could define new abstractions (e.g. define new functions or selectors).

Based on this observation, I phrased the following research questions about language design:

> What would CSS be like if it enabled the style sheet author to define new abstractions, similar to those that the CSS specifications define (functions, selectors, shorthand properties, named colors, pseudoelements and media queries)? Would authors like to have these features? Would these features be enough to consider this extended CSS language a full-featured programming language, or authors would like to have other features as well?

As I will show in Section 4, web developers generally gave a positive feedback about extending CSS in this direction. However, they would also like to use some other abstraction features as well.

Apart from the usefulness of a such extended CSS language, its **implementability** must also be considered. The FunCSS language should have an implementation that works in all of the currently widely used web browsers, without requiring visitors to install any additional software.

The implementation should also be **efficient** enough that this language can be used in **animations**. This would require the calculations involved in the implementation to be fast enough, so that an acceptable frame rate can be achieved in the animation.

When using CSS, all changes to the document or the environment (e.g. a new element is inserted into the document or the window is resized) are automatically reflected in the presentation, with all necessary recalculations and re-renderings automatically done by the web browser. As such, CSS follows the **reactive programming** paradigm [3]. I consider this a substantial feature of CSS, because it helps style sheet authors largely. Otherwise, authors would need to take extra care for updating styles after a change occurs in the document. Similarly, my extension of CSS should follow the reactive paradigm.

Based on these requirements, I phrased the following research questions about the implementation:

> How can the extended CSS language be implemented so that it can be used in most widely available web browsers without installing additional software? Can it be implemented efficiently enough so that it can be used for developing animations? Can it be implemented so that the reactive nature of CSS is retained?

The answers to these questions is not trivial. Implementing the reactivity of CSS can be complicated, and one might expect that the implementation cannot be efficient enough for animations. Also for the reactivity, the language runtime needs to react to changes in the document. Listening to changes in the document might not be supported in all browsers.

In early 2014, I used the Meteor web development framework[1] in a web development project. The Meteor framework contains a library, called Tracker, which provides an implementation of functional reactive programming in JavaScript (I will present Tracker in 2.2.1). While using Tracker, I realised that the paradigm that Tracker uses is fairly similar to the reactive paradigm of CSS. I also experienced that Tracker is fast enough for certain animations. Based on these observations, I supposed that my planned extension of the CSS language could be implemented with Tracker or with a similar library.

In early 2015, I started to actually design my planned extended CSS language. The definition of functions became a substantial feature of the language, so I named my language *Functional CSS*, whose abbreviation is *FunCSS*.

1.3 Achievements of the thesis

My achievements presented in this thesis are the following.

- I conducted a survey among web developers about their requirements of CSS-based languages.
- I designed syntaxes for defining functions in FunCSS based on the results of the survey.
- I developed a static type system for FunCSS based on the CSS specification
- I made a prototype implementation of FunCSS

1. http://www.meteor.com/. Accessed: 15-Jun-2015

1.3 Achievements of the thesis

Section 2
Related work

2.1 Adding abstractions to CSS

CSS is a simple language and it does not have many abstraction features. For this reason, many developers have created solutions to add abstractions to CSS. The solutions have a wide spectrum, I will present here six categories

2.1.1 CSS methodologies

Several methodologies have been developed for creating abstractions in CSS. These methodologies deal with standard CSS code, and make some restrictions about how standard CSS should be used. This means that authors use abstractions that are not part of the CSS language, but are created with using conventions and existing features of CSS. For example, *Object-Oriented CSS*,[2] uses the notion of subclassing, and implements it by adding both the superclass and the subclass to `class` attribute of the element. Other examples

2. http://www.stubbornella.org/content/2009/02/28/object-oriented-css-grids-on-github/. Accessed: 15-Jun-2015

of CSS methodologies are *SMACSS* [3], *BEM*[4], *SUIT CSS*[5], *Atomic CSS*,[6] *Organic CSS*,[7] *Enduring CSS*.[8].

2.1.2 CSS libraries

Another category of tools that can help work with CSS is CSS libraries. Some examples are Bootstrap,[9] Pure[10] and Cascade Framework[11]. These are collections of CSS rules that can be added to style sheets to help in styling. All of these mentioned libraries define components like panels, menus, buttons etc, which an be considered abstractions.

2.1.3 Debate about CSS variables

Variables have been proposed to be added to CSS in 1998 [4]. In 2008, Daniel Glazman and David Hyatt published a proposal about how variables could be implemented. In the same year, Bert Bos, one of the original designers of CSS, opposed the idea of variables [5]. Still, in 2012, the idea of Glazman and Hyatt was developed into a W3C Working Draft. The latest draft of this specification dates from May 2015 [28].

2.1.4 Generating CSS on the server side

Bos [5] mentions several working solutions for generating CSS on the server side, by using PHP or SSI. These techniques involve adding substituting variables with actual values in the generated CSS code.

3. http://smacss.com/. Accessed: 15-Jun-2015
4. https://en.bem.info/method/. Accessed: 15-Jun-2015
5. http://suitcss.github.io/. Accessed: 15-Jun-2015
6. https://github.com/nemophrost/atomic-css. Accessed: 15-Jun-2015
7. http://krasimir.github.io/organic-css/. Accessed: 15-Jun-2015
8. http://benfrain.com/enduring-css-writing-style-sheets-rapidly-changing-long-lived-projects. Accessed: 15-Jun-2015
9. http://getbootstrap.com/. Accessed: 15-Jun-2015
10. http://purecss.io/. Accessed: 15-Jun-2015
11. http://www.cascade-framework.com. Accessed: 15-Jun-2015

2.1.5 CSS preprocessors

CSS preprocessors are applications that take a file in a specific language as an input and produce CSS as an output. The three most popular CSS preprocessors are *Less*,[12] *Sass/SCSS* and *Stylus* [6]. Sass and SCSS are actually two syntaxes of the same preprocessor, with the same semantics. All of these three preprocessors have several abstraction features, here I will present three features that all of them support.

Variables can be used in all of the three major preprocessors. As an example, in SCSS, one can define a variable and use it as follows:

```
1 $x: 20px;
2 body {
3   margin: $x;
4 }
```

Nesting CSS rules means that a CSS qualified rule can be placed into another. The selectors of these are then combined. The following code have the same meaning in Less, SCSS and Stylus.

```
1 body {
2   background: black;
3   > .container {
4     background: white;
5   }
6 }
```

It is compiled to the CSS code

```
1 body {
2   background: black;
3 }
4 body > .container {
5   background: white;
6 }
```

Extending CSS rules means that a CSS qualified rule is extended with declarations from another. The following example is SCSS code. Line #5 instructs the preprocessor to copy line #2 into the second rule (this is only true conceptually, technically it is solved somewhat differently).

12. http://lesscss.org/.

```
.button {
  background: #gray;
}
.danger-button {
  @extend .button;
  color: red;
}
```

2.1.6 Calculating style on the client side

JavaScript is a full-featured programming language which is available in all modern web browsers. Through the CSS Object Model [23], JavaScript can access the style sheets of the page, and style properties of individual elements, and set them arbitrarily.

There are several JavaScript libraries that ease the work with styles. *jQuery*, *Prototype* and *React.js* are three examples. All of the three provide a simple way to set style properties of elements using a JavaScript data structure.[13] By using JavaScript data structures to store style data, one can use the abstraction features available in JavaScript. A good explanation of this technique can be found in [7] by Facebook developer Christopher Chedeau.

JavaScript code can be also used to create animations. JavaScript in web browsers can use timers to execute scheduled actions, so it can by repeatedly update CSS style properties of elements, and thus create animations. Several JavaScript libraries have support for animations, with special handling of CSS values (jQuery,[14] d3.js[15]).

2.2 Functional reactive programming on the web

Reactive programming is a paradigm that models the application as a data flow graph, and data as values that change over time (usually called behaviors or signals). Functional reactive programming is a variation of the reactive paradigm, commonly used in functional programming languages [3].

13. http://api.jquery.com/css/,
http://api.prototypejs.org/dom/Element/prototype/setStyle/,
https://facebook.github.io/react/tips/inline-styles.html Accessed: 15-Jun-2015
14. http://api.jquery.com/animate/ Accessed: 15-Jun-2015
15. https://github.com/mbostock/d3/wiki/Transitions Accessed: 15-Jun-2015

Several implementations of functional reactive programming have been developed for the web. Some of them use JavaScript (like Bacon.js[16] and ReactiveX[17]) while others use a dedicated language (like Elm[18]).

Below is an example from the website of Elm, which displays the mouse position continuously as text.

```
main : Signal Element
main =
  Signal.map show Mouse.position
```

Here the program takes `Mouse.position`, a signal, and maps the `show` function on it, and this will be the result.

2.2.1 The Tracker library

The Tracker library is a part of the Meteor web development framework. It is an implementation of functional reactive programming in JavaScript. However, contrary to other FRP libraries like Bacon, ReactiveX or Elm, it does not handle streams explicitly like in the above example. Instead, streams are represented with JavaScript functions. Those functions that represent streams are called **reactive data sources** in Tracker. When these are normally called, only return their current value. However, when called within a special context, called a **reactive computation**, reactive data sources will notify Tracker to re-run the whole computation when their value changes.

The above example, a text that always shows the mouse position, would look like this with Tracker:

```
Tracker.autorun(function() {
  element.innerHTML = getMousePosition();
});
```

`autorun(…)` serves for creating the reactive computation. It will call the given function once. After that, whenever the `getMousePosition()` function signals that the mouse position has changed, it will call the given function again.

I used the Tracker library to implement the runtime environment of FunCSS. I found that Tracker is especially appropriate for this purpose, as the paradigm of Tracker is very similar to the paradigm of CSS.

16. https://baconjs.github.io/ Accessed: 16-Jun-2015
17. http://reactivex.io/ Accessed: 16-Jun-2015
18. http://elm-lang.org/ Accessed: 16-Jun-2015

2.2.1 The Tracker library

Section 3
An analysis of CSS

My goal while designing the FunCSS language was to make it compatible with CSS, so that any CSS code should be a valid FunCSS code with the same semantics. I also sought for compatibility in terms of design principles and programming paradigms, i.e. the principles of the FunCSS language should be similar to those of CSS. In order to create a language that has a good compatibility with CSS, first I needed to thoroughly analyse CSS itself.

In this section, I present the CSS language from a language designer's point of view. First, I present some powerful features of CSS that are rarely found in popular programming languages. Then I present the core syntax, the selectors and the property values of the CSS language.

3.1 Some rare "gems"

CSS is not meant to be Turing-complete, so it is not a programming language in the common sense of the word. However, it has several algorithmic features that help style sheet authors to express their (sometimes complex) intentions about the appearance of the document. Many of these features can be found in general-purpose programming languages, however, some of them are quite rarely used despite being fairly powerful:

Selectors. One powerful algorithmic feature of CSS is the system of selectors, a construct for selecting a subset of the elements of a web page, based on their attibutes or hierarchical structure.

Cascading and inheritance. The selector system is complemented by the system of cascading and inheritance, which defines how conflicting style declarations are resolved. It is based on a heuristics that estimates the specificity of selectors.

Physical units of measure. An important tool is physical units of measure and percentages, which are natively handled by the type system. These can help the author to express values in a safe and concise way.

Reactivity. A fourth important feature of CSS is reactivity, which means that changes in the web page or the style sheet are automatically updated in the appearance of the page, without the need for manual update logic.

When designing FunCSS, one of my goals was to keep these useful features of CSS, and extend them if needed.

3.2 The core syntax

The CSS style sheet consists of rules: qualified rules and at-rules. **Qualified rules** are the more common kind of rule, they describe the style properties of a chosen set of document elements. Here is an example style sheet, with a single qualified rule, and the example rendering beside.

Code 3.1
A qualified rule

```
.title {
  opacity: 1;
  border: 1px solid green;
  font-family: "Quattrocento";
  font-size: 180%;
  transform: rotate(4deg);
}
```

This qualified rule contains a selector (`.title`) and a list of declarations within `{…}`. The selector is used to select a set of elements of the document for which the rule applies. Selectors have their own grammar, detailed in Section 3.3.

The list of declarations consists of declarations separated by semicolons. A declaration consists of a property name followed by a colon (`border:`) and a corresponding value. The possible set of values depends on the property, but it may include numbers (`1`), lengths (`1px`), keywords (`solid`), colors (`green`),

strings (`"Quattrocento"`), percentages (`180%`), angles (`4deg`), functional notations (`rotate(4deg)`), and combinations of these. I talk about property values more in detail in Section 3.4.

Another type of rules are **at-rules**. These are usually special instructions for the web browser. They start with an at-sign (`@`), directly followed by a keyword, and they might or might not have a *prelude* and a *body*, depending on the kind of at-rule. The following example contains two at-rules, one instructs the web browser to include another style sheet, and the other defines the margins of the page when the web page is printed. The first one has a prelude up to the semicolon, the second one has a body inside the curly brackets.

Code 3.2 At-rules

```
1 @import "other_file.css";
2 @page {
3   margin: 3px;
4 }
```

To achieve forward compatibility, all versions of CSS share a common grammar, called the **core grammar** [19, s4.1.1], which describes the basic structure of CSS code, including rules, declarations, basic value types, blocks and functional notations. It does not restrict the contents of property values, selectors, at-rule preludes and at-rule bodies.

After parsing the input with the core grammar, web browsers need to continue the parsing with different grammars for the different parts of the input. They parse selectors with the Selector Grammar, property values with the property value grammar corresponding to the specific property, and at-rules with the grammar corresponding to the kind of the at-rule. In this thesis, I call these grammars the **subgrammars** of CSS, and the corresponding languages the **sub-languages**.

3.3 Selectors

The CSS language has a powerful sub-language for selecting a set of elements from the web page. This language, the Selector Language is used in preludes of qualified rules to define the set of rules for which the qualified rule applies.

$$\text{expression} \times \text{element} \to \text{boolean}$$

that is, given a selector expression and an element, it tells whether the element matches the selector or not [20].

There are two kinds of selectors: simple selectors and contextual selectors [8, s6.2.1]. Simple selectors only take the type and the attributes of the specific element into consideration, while contextual selectors also consider the place of the element in the DOM tree. Some of the simple selectors are listed in Table 3.3:

Table 3.3 Simple selectors [20]

Pattern	Name	Description
`*`	Universal selector	Matches any element
`h1`	Type selector	Matches an element with type `h1`
`#xyz`	ID selector	Matches an element with ID `xyz`
`.asdf`	Class selector	Matches an element with class `asdf`
`:hover`	Pseudoclass selector	Matches an element with the mouse pointer above
`:disabled`	Pseudoclass selector	Matches an element in a disabled state

Simple selectors can be grouped into sequences, like `#xyz.asdf` or `input.asdf:hover:disabled` [20, s4]. Sequences of simple selectors may contain at most one universal selector or type selector, as the first element.

Sequences of simple selectors can be combined by *combinators*. Each combinator defines a requirement for two elements in the DOM tree, in addition to the requirements posed by both sequences of simple selectors.

Table 3.4 Combinators [20]

Pattern	Name	Description
`E F`	Descendant combinator	Element matched by selector F must be a descendant of an element matched by selector E
`E > F`	Child combinator	Element matched by selector F must be a child of an element matched by selector E
`E + F`	Adjacent sibling combinator	Element matched by selector F must be the next sibling of an element matched by selector E
`E ~ F`	Next sibling combinator	Element matched by selector F must be any of the sibling of an element matched by selector E after the element itself

If two rules are conflicting, one is chosen based on an algorithm described in the specification, called the **cascade** [27]. The cascade applies several principles. First, the **origin** of the rule is considered: does it come from the browser defaults, the author of the web page, or the user? (The user's ability to override the appearance of web pages was considered an important feature

in the initial time of the web [1], however, today it is not commonly used [8, s6.5.1.2].)

If the origin does not decide the order of two rules, a heuristic is used to make the more specific rules stronger: the **specificity**. The specificity is a tuple of numbers which is calculated from the number of different syntactic elements in the selector, then specificities are ordered lexicographically [27]. Finally, if the specificity of two rules are the same, they are ordered in their **order of appearance**: rules that come later are stronger.

3.4 Property values

Possible values of properties are lists of **component values**. Component values include values of basic types (keywords, strings, urls, numbers, dimensions etc.) and structured values (functional notations and blocks).

Each standard property has restrictions on what values are allowed for it. The CSS standard defines a grammar for each property, which defines the allowed values for the particular property. For defining the grammar of a property type, the standards use a dedicated syntax, called *Value Definition Syntax*, described later in this section.

3.4.1 Component value types

Table 3.5 shows a list of the component value types that are allowed in CSS. Intuitively, all integers are as well numbers. Dimension has five subtypes: length, angle, time, frequency, and resolution. The number `0` can be used in any dimension subtypes. Percentages are usually interpreted as a portion of another value (number or dimension), depending on the context in which they are used.

Functional notations and blocks contain another list of component values. Blocks are usually not directly used in property value definitions, however, e.g. the definition of the `calc(…)` functional notation allows parenthesised blocks in its argument [25].

Delimiters do not have an intrinsic value, but they are important in the syntax of several properties and functional notations. Whitespace is ignored in all standard property values, apart from the fact that it can separate two tokens that otherwise would be considered one token (e.g. `2px - 1px` are five tokens:

Table 3.5 Component value types [25]

Name	Examples
Textual types	
keyword	`auto lower-roman`
string	`"Hello World!" 'Times New Roman'`
url	`url(http://funcss.org/)`
Numeric types	
integer	`-5`
number	`3 -2.5e-2`
dimension	`3px 90deg 0.5s 230Hz 96dpi`
percentage	`75%`
color	`#ffcc00`
Structured types	
functional notation	`translate(20px, 0)`
blocks	`(…) […] {…}`
Miscellaneous	
delimiters	`, /`
whitespace	space, newline, tab

two dimensions, two whitespaces and one delimiter, while `2px-1px` is a single dimension token with the invalid unit `px-1px`).

3.4.2 Value definition syntax

CSS specifications use a syntax, the Value Definition Syntax (VDS), for defining the allowed values of properties. This syntax is capable of defining any regular grammar on component value lists. Table 3.6 shows the elements of the VDS syntax.

The specification [25] does not mention *constant number* explicitly, but it uses it in the definition of `font-weight` [19, s15.6], so I included it in the table.

Combinators need to be interpreted as follows: several juxtaposed types mean that all of these values need to be present, in the given order. Several types

Table 3.6 Elements of the CSS Value Definition Syntax [25]

Name	Example
Terminals	
Keyword	`auto`
Constant number	`400`
Reference to a basic type	`<number>`
Reference to a type defined elsewhere	`<relative-size>`
Reference to a type of a property	`<'border-color'>`
Delimiter	`,`
Structures	
Functional notation	`rgb(<number>, <number>, <number>)`
Multipliers	
Zero or one	`<number>?`
Zero or more	`<number>*`
One or more	`<number>+`
In a range	`<number>{2,4}`
One or more, separated with commas	`<number>#`
Grouping	
Grouping	`[…]`
Combinators	
Juxtaposition	`<number> <length>`
And	`<length> && <number>`
Inclusive or	`<number> || <length>`
Exclusive or	`<number> | <length>`

combined with *and* (`&&`) means that all of the values need to be present, but in any order. *Inclusive or* (`||`) means that one or more of the given values need to be present, in any order. *Exclusive or* (`|`) means that exactly one of these needs to be present [25, s2.2]. Whitespace is allowed between the values with any of the four combinators.

3.4.2 Value definition syntax

Section 4
Designing the FunCSS language

4.1 The design goals

Based on the research questions in 1.2, I phrased the following design goals for FunCSS:

a) Declarative (rule-based and reactive). Similar to CSS, FunCSS should also be a declarative language, and it should follow the rule-based and the reactive programming paradigms.

b) Human-friendly syntax. CSS has been designed to have a human-friendly syntax [8, s7.2.3.3, 9, 21, s5]. The syntax of FunCSS should also be optimised for ease of use.

c) Fast. FunCSS should be fast enough for animations.

d) Programming abstractions. FunCSS should contain sufficient support for creating abstractions, to ease everyday web development tasks, and to avoid code repetition.

e) Browser compatibility. FunCSS should work identically on all currently widely used browsers.

f) Full access to the document. CSS supports some styling based on the current structure and status of the document elements, but not arbitrarily. To consider FunCSS a full-featured language, it should be able to access all data of the document

g) Turing-complete. CSS cannot be used to compute an arbitrary calculation. FunCSS, as a full-featured programming language, should be able to compute any algorithm.

4.2 Empirical research

4.2.1 A text mining study on the demand for CSS preprocessor features

In order to see what developers require from an extended CSS language, I analysed comments of a blog post[19] which compares the two most popular CSS perprocessors, *Sass* and *LESS* with each other. I analysed the first 170 comments in order of appearance on the page. I identified arguments that commenters gave for or against one preprocessor or another. I did not consider which preprocessor they argued for, only the argument. I categorised these arguments into 12 categories. The results can be seen in Table 4.1.

When interpreting then numbers in Table 4.1, I took into consideration that the blog post compared two preprocessors, which are similar in many respects. Thus, debate in comments focused more on the differences between the two preprocessors and not on their common strengths. As such, these data cannot

19. https://css-tricks.com/sass-vs-less/. Accessed: 26-May-2015.

Table 4.1 What preprocessor users consider important. Results of text mining in comments of a blog post

Category	Number of posts
easy workflow (easy installation, compilation, debugging, platform independence, easy continuous integration, easy automated build)	40
availability of good libraries (Compass (10), Bootstrap (4), others (11))	25
succinct syntax (concise, easy to remember, much expressive power, do more with less)	23
compatibility with a specific development platform (Windows support, Mac support)	15
browser compatibility (auto-prefixing, auto-workarounds, JavaScript is not needed)	14
runtime performance (optimizations, spriting)	8
safe (not whitespace-sensitive, misspellings are easily recovered, safety-oriented library structure, linting support)	7
easy to learn (memorable constructs, good documentation, good tutorials)	6
compatibility with CSS (with current version, with future versions)	6
community (nice website, many users)	4
ergonomic syntax (easy to read, convenient, conventional)	3
software quality (support, maturity, well-defined scope)	3

be considered an exact source of information for measuring the relative importance of features. Commons strengths of Less and Sass are more likely to have less relative frequency compared to their relative importance.

Nevertheless, I could make several qualitative observations, that served as a good basis for a later survey, which I will present in 4.2. An observation that I made was that *succinct syntax* had relatively many mentions (23), despite it is an advantage of both of the preprocessors. This means that the succinctness of the syntax is quite important.

Another observation is that browser compatibility has also had a significant number of mentions (14). From this, I concluded that I should focus more on browser compatibility features in FunCSS.

It was surprising for me that the Compass framework was mentioned in relatively many (10) comments. Compass is a framework written in the Sass language, and its main features include browser compatibility features and automatic spriting.[20] From this I concluded that automatic spriting might be

an important feature, and that, again, browser compatibility is of high importance.

To have better measurements on these features, I included some of them into the survey that I explain in the next section. See Appendix B Question 7 for the results on these features.

4.2.2 A survey about planned features

I conducted an online survey between 28 Apr and 26 May 2015 about the planned features of FunCSS. The link to the survey was posted on 28 Apr to two Meetup Groups: *BudapestJS* and *Budapest Frontend Meetup*. I used the mailing list of the former group and the online forum of the latter, because the latter did not have a mailing list. The two meetup groups nearly the same number of members (1286 and 1352 respectively), but my post to the mailing list of BudapestJS probably reached much more people than my post to the Budapest Frontend Meetup forum (I received three replies on the mailing list and I did not receive any replies on the forum).

I received 32 submissions for the survey. The survey results are in Appendix B. In the survey, I asked respondents 8 questions. In each question, respondents had to specify their feeling about some items. There were altogether 42 items in the questions: 36 items in the first 7 questions referred to possible features of FunCSS, and 6 items in the last question referred to possible revenue models.

I used a custom-made slider for each item to input numerical data from the respondents. The button of the slider was an animated emoticon that changed its facial expression as the user moved the slider. I asked respondents to set the sliders so that it best matches their feeling about each item. These were the possible facial expressions (except the first one, which indicated the skipped items), distributed evenly on the slider:

For aggregating the input data, I used quartiles. Another simple option would have been using averages, by averaging the coordinate of the button on the slider. However, the scale was not genuinely an interval scale, but rather an ordinal scale, because the facial expressions could have been distributed with

20. Spriting is an optimisation technique which involves collating several images into a single one, to save download time.

any other distribution on the slider. So I assumed that using averages could be misleading.

I included the first and third quartiles in my analysis because I wanted to compare the distribution of the answers. To compare the distributions of two items, I used the inter-quartile ranges (IQR). With using IQRs, I contradicted my previous decision that I should not use aggregations that depend on the differences between values. However, I only used IQRs for fairly vague judgements, like "this question is very controversial" or "there is large consensus in this question", and I did not decide for or against a feature based on the IQRs. With these restrictions, I think using IQRs was appropriate. I did not include Q0 and Q4 in the analysis, because they are sensitive to outliers.

For comparing answers within a category, I defined a partial ordering between the answers. One answer is at least as good as the other if it scored at least as well in Q1, Q2 and Q3. That is,

$$\begin{aligned} \text{item}_1 \precsim \text{item}_2 \Leftrightarrow\ & \text{Q1}(\text{item}_1) \leq \text{Q1}(\text{item}_2) \wedge \\ & \text{Q2}(\text{item}_1) \leq \text{Q2}(\text{item}_2) \wedge \\ & \text{Q3}(\text{item}_1) \leq \text{Q3}(\text{item}_2). \end{aligned}$$

If the relation does not hold in the other direction, then item_2 is better than item_1 ($\text{item}_1 \prec \text{item}_2$). If an item is at least as good as all other items among the items in the same question, then we call this the best item (there can be several best items).

4.3 Type system

I decided to make FunCSS statically typed. One reason for this is that static typing can help avoid several errors in the code in compile time. Another reason is that several values in FunCSS (just like in CSS) do not have an intrinsic type, but their type depends on the context in which they are used. For example, the value `red` can be a color or a counter name, depending on the context. To know the exact meaning of values, I chose incorporate static typing into the compiler.

As far as I know, CSS does not have a formalised type system. However, its specification defines a set of basic types [25] (see 3.4.1) and the concept of *component values* which include all values of basic types, functional notations and two delimiters (`,` and `/`). For technical reasons, some other token types are also allowed in component values (like blocks, which are not allowed in

property values, or whitespace tokens, which are ignored). For the discussion of the type system I will ignore these additional possible values.

The specifications of CSS use a syntax, the Value Definition Syntax (VDS) [25], for defining the allowed values of properties, and the allowed arguments of functional notations. The VDS defines regular grammars over lists of component values. See 3.4.2 for an introduction about the VDS.

I designed a type system for FunCSS based on the VDS. Currently the FunCSS type system (FunCSSts) is basically a reinterpretation of the grammatical rules of VDS as a type system. I consider nonterminals of grammars described with VDS to be types, and the generated sentences to be expressions. As an example, `<number>` is a nonterminal in VDS that generates the `0`, `1`, `0.1` etc. tokens. In FunCSSts, `<number>` is a type whose inhabitants are `0`, `1`, `0.1` etc. Similarly, `<number> || [yes|no]` (which means a number and/or a `yes` or `no` keyword, in any order) is a type in FunCSSts and the component value lists `3`, `3 yes`, `no 3` and `no` are all inhabitants of this type. As a general rule, if a VDS grammar *G* generates a sentence α, then α inhabits the type represented by *G*.

In Appendix A I included a premature formalism for a subset of the FunCSSts type system described here. Here I give some comments about FunCSSts.

As FunCSSts is basically a reformulation of a regular expression system as a type system, it has some unusual properties. One unusual property is that every term inhabits an infinite number of types. For example, the term `normal` inhabits all of the following types:

- `normal`
- `[normal | solid]`
- `[[normal | solid] || <length>]`
- `[[normal | solid] || [<length> && <number>]]`
- etc.

Even some single-element terms belong to several basic types. For example the term `0` belongs to the `<number>`, `<integer>`, `<length>`, `<angle>`, `<time>`, `<frequency>`, `<resolution>` types. Similarly, the term `red` belongs to the `<color>`, `<custom-ident>` and `red` types (the latter is a keyword type, having only one inhabitant).

As every term inhabits infinitely many types, the type of an expression cannot be inferred uniquely without additional information about the context of the term. When used in a property value, however, the property value constrains the type of the value. For example, the declaration

```
1 background: red;
```

constrains `red` to the value type of the `background:` property, which is something like `<color> || <bg-image>` (in reality it is more complicated). Then, it can be inferred that `red` is a `<color>` and is not a `<bg-image>`, so this lets the type checker uniquely infer the type of `red`. In the following sections, I call this process **checking** `red` **against** the type `<color>`.

4.4 A database of CSS properties

As I have shown in the previous section, the type of a component value in a property cannot be inferred without knowing the value type of the property that contains it. This has a serious implication: the FunCSS compiler has to know the types of all properties that can be used in CSS. This means that either the compiler or the standard library should contain the definition of all standard CSS properties.

Maintaining an up-to-date database about all CSS properties is a difficult task. However, a such database in FunCSS can have several benefits. For example, if the author mistypes a property name, or enters a property value of a wrong type, the compiler will probably notice it and signal an error.

4.5 Function definitions

One of the core features of FunCSS is the ability to define functions. There are two kinds of function definitions: one where the function body is a FunCSS expression, and the other is where the function body is a JavaScript code block. I discuss the latter in 4.8.0. In this section, I discuss functions whose body is defined by a FunCSS expression: the syntax, the type annotation syntax, and the semantics of these definitions.

4.5.1 Syntax

I had several ideas for the syntax of function definition. I considered all of the following. (All these examples would define a function `f(…)` that return the sine of this argument. Type annotations are removed from these examples for simplicity):

```
1  @function f($x) sin($x);
2  @function f($x) = sin($x);
3  @fun f($x) sin($x);
4  @fun f($x) = sin($x);
5  @def f($x) sin($x);
6  @def f($x) = sin($x);
7  @number f($x) sin($x);
8  @number f($x) = sin($x);
9  f($x) = sin($x);
10 f($x) : sin($x);
```

In the survey that I conducted about syntaxes, I included four of these examples (lines 1, 3, 4 and 9) to understand how web developers think about them (Appendix B Question 3). As a result, I got line 9 to be the single best of the four. So based on the survey, I chose `f(x) = sin(x)` as the syntax for defining functions.

Line 10 is similar to line 9, and I have not asked line 10 in the survey. However, in Question 2, four out of 32 respondents suggested to use the = sign instead of the : sign for assigning values to variables. As defining a function is similar to defining a variable, I chose the = sign defining functions, too.

4.5.2 Type annotations

Another reason why the = sign is better than the : sign for specifying the value of a function, is that : can be used for type annotations. For example, the above definition with type annotations looks like this:

```
1 f($x:number):number = sin($x);
```

The syntax of these type annotations are based on the syntax of TypeScript.[21] TypeScript is a statically typed extension of JavaScript.

As some people have suggested in the survey, type annotations should not be mandatory if they are unambiguous from the context. For example, the return type of the following function can be inferred to be `integer`:

```
1 f($x:number) = 5;
```

The exact semantics of the type inference is yet to be researched.

21. http://www.typescriptlang.org/. Accessed: 15-Jun-2015

4.5.3 Semantics

Currently the semantics of the

```
f($x:number) : number = sin($x);
```

function definition is that it extends `<number>` type with new possible value: functional notations with the name `f` and a number as an argument. In other words, it redefines `<number>` as follows

let `<number>` be `f(<number>) | <number>'`

where `<number>'` is the original `<number>` type before the redefinition. Now when `f(5)` is checked against the `<number>` type, the first branch will match. Then, `sin(…)` will be substituted, with the argument properly passed.

This semantics has the implication that functions can be **overloaded**. For example, the declarations

```
f($x:number) : number = abs($x);
f($x:angle) : number = cos($x);
```

will add two new branches to the `<number>` type. When a term is checked against the `<number>` type, both of the branches will be considered. So the term `f(-4)` will calculate `abs(-4)`, and `f(90deg)` will calculate `cos(90deg)`. However, `f(0)` is ambiguous: it will raise a typing error.

Furthermore, the semantics of function definitions also implies that functions can be **overloaded** based on their **return types**:

```
g($x:number) : number = abs($x);
g($x:number) : angle = atan($x);
```

These declarations extend both the `<number>` type and the `<angle>` type, with one branch each:

let `<number>` be `[g(<number>) | <number>']`
let `<angle>` be `g(<number>) | <angle>'`

where `<number>'` is the previous value of `<number>` and `<angle>'` is the previous value of `<angle>`.

Note that the two branches added to the above two types usually do not contradict. When the term `g(5)` is checked against the `<number>` type, the first definition is used, and the result will be `abs(5)`. When the term `g(5)` is checked

against the `<angle>` type, the second definition is used, and the result will be `atan(5)`.

As an example, `g(5)` in the following two property declarations evaluate to two different values.

```
opacity: g(5);
transform: rotate(g(5));
```

However, when `g(…)` is used in a situation where both a number and an angle would be allowed, the choice of the definition would be ambiguous. I do not know about a CSS property which accepts both a number and an angle. However, the above defined `f(…):number` function can accept both a number and an angle as its argument. Thus `f(g(5))` interpreted as a number would mean a type error because of the ambiguity of `g(…)`.

4.6 Preprocessor integration

The survey results have shown that some common preprocessor features, namely nesting CSS rules and extending CSS classes (see 2.1.5) are highly valued by web developers (Appendix B Question 7). Based on the results, one can anticipate that many developers would be less likely to choose FunCSS if it did not contain these features.

Writing a preprocessor is a huge task. The three major preprocessors, Sass, Less and Stylus have been being developed since 2006, early 2010 and late 2010 respectively, and each one has 100 to 200 contributors (according to their repositories on GitHub). This shows that a huge amount of work has been invested into each of them. If I wanted to replicate the features of these preprocessors in FunCSS, I would need to invest much work into it as well, and it could possibly result in a software of lower quality than these popular preprocessors.

Instead of replicating the functionalities of the preprocessors in FunCSS, therefore, I decided to integrate a preprocessor into the compilation process. However, this is not implemented yet.

There are several possible scenarios to implement preprocessor integration. First, to integrate with only one preprocessor, and make the preprocessor features part of the language specification. This would be the simple for the user of FunCSS, because no configuration would be necessary. However, the language specification would depend on a third-party software. The second

scenario would be to integrate FunCSS with one preprocessor only, but make it optional. This would be easier for the purpose of language specification, however it could be more complicated for the users of FunCSS. Finally, a possible scenario is to integrate with all three major preprocessors. This would have the advantage that FunCSS could be used together with the user's preferred preprocessor. However, the integration with all three preprocessors would be costly to implement and maintain. Therefore, this topic requires further research.

4.7 Variables

Throughout this thesis, I use the word *variables* when referring to the abstraction where a name represents a value. However, there are several different variations of the semantics of variables or similar constructs. In imperative programming languages, variables usually represent slots of memory whose content can be read and written. In functional programming languages, variables usually represent function arguments, and can only be read. In FunCSS, I have not yet decided about the semantics of variables, only about their syntax.

Originally, I did not plan to include variables in FunCSS. However, the results of the survey that I made showed that variables one of the most demanded constructs that developers would like to see in FunCSS (Appendix B Question 6). The large demand for the support of variables is not surprising if we consider that most programming languages support them.

The decision about the variable syntax was not easy. The three major CSS preprocessors use three different syntaxes for variables. Sass uses `$x`, LESS uses `@x` and Stylus uses `x` (but also permits `$x`). All three syntaxes have advantages and disadvantages. `$x` does not interfere with any CSS construct (CSS does not have similar constructs), while `x` interferes with CSS keywords and `@x` interferes with CSS at-keywords. There are hundreds of keywords in CSS, while there are only a dozen at-keywords. Therefore, the risk of accidental collision is lower with `@x` than with `x`. However, `x` is easier to type, and might be easier to read.

There is a planned feature in CSS that is similar to variables, however, with somewhat different semantics [28]. The current plans for the syntax of CSS variables is `--x` for declaration and `var(--x)` for use. This syntax does not interfere with any previous construct in CSS, but is more difficult to type, and might be more difficult to read than `$x` (`var(--x)` contains 7 additional

characters besides the name). Originally, the designers of the CSS Variables specification proposed `$x` for this feature, but they changed the syntax later [10, 11].

I included `$x`, `@x` and `x` in the survey (Question 1), the three variations used by the three most popular CSS preprocessors. I also added a fourth syntax, `%x`, which CSS does not use for other purposes. In the results, `$x` was the single best option.

Based on the results, I finally chose `$x` to be the syntax for variables. One exception is in the argument of literal JavaScript functions (which I will present in the next section). There variables cannot collide with CSS constructs, so `x` can be allowed.

I have not yet decided about the semantics of variables. The three major CSS preprocessors (Sass, Less, and Stylus) use a semantics similar to what imperative languages have, and this could be an option. Another option is to implement the semantics of the CSS Variables draft, and use `$x` as a syntax for it (as it was originally proposed). A third option is to make variables have the same (or similar) semantics that functions without arguments have. This question is to be decided based on further research.

4.8 Integration with JavaScript

In order to let style sheet authors define arbitrary functions, we need a way to describe an arbitrary algorithm in FunCSS. One option is to provide constructs in FunCSS (e.g. built-in functions for conditionals, recursion) that are capable of defining any algorithm. Another option is to let the author write JavaScript code and access it from FunCSS code. The first option, using built-in constructs, is a topic that needs further research. In my current research, I analysed the integration of JavaScript code into FunCSS.

Using JavaScript has several advantages and disadvantages. An advantage is that most front-end web developers are familiar with JavaScript to at least some extent. This means that if FunCSS supports JavaScript code, the learning curve for front-end web developers will be shorter. A disadvantage is that JavaScript code can have side effects, and thus it can spoil the declarative nature of FunCSS. However, we can make some conventions which, if the author obeys them, can ensure the purity of the code.

I want FunCSS to be easy to learn for web designers and developers, so I chose to integrate JavaScript with FunCSS. I do not consider the possible side

effects in JavaScript codes to be a problem. Side effects in reactive code can be exploited cleverly by authors. For example, a JavaScript code could start an asynchronous data retrieval whenever some data are requested by FunCSS code, and restart the computation when the requested data arrives. So I chose to let the author use JavaScript in FunCSS.

There are several ways how JavaScript can be integrated into FunCSS. An obvious one is to let FunCSS functions can be implemented in JavaScript. Another possibility is that custom selectors can be implemented as JavaScript functions which take an element as an argument, and return a boolean value that tells if the element matches the selector. JavaScript code could be used as well to define units, media queries, or even colours, however, these topics require further research. In the current version of FunCSS, I implemented support for functions with JavaScript definition.

4.9 Embedding literal JavaScript

A decision that I had to make was whether to support the embedding of literal JavaScript code. If JavaScript is embedded literally, it might make the code easier to read and understand. Otherwise,the author would need to look for the definition in separate files. However, embedding literal JavaScript might as well confuse some readers, as a web developer noted in an interview I made. For now, I chose to let authors embed literal JavaScript. Later, I might add a possibility to use non-embedded JavaScript code too.

Embedding literal JavaScript code into a CSS-like language is not trivial as far as syntax is concerned. The lexicon of JavaScript is not compatible with that of CSS. For example in CSS, `u+30` and `U+030` are equivalent Unicode range tokens, while in JavaScript these are two different expressions (additions with two different values on both sides). Also in JavaScript, `/}/` is a single regular expression token, while in CSS, `/}/` contains a closing bracket token, which influences block structure.

It is possible to embed JavaScript code into a FunCSS block or value, as long as the JavaScript code does not contain lexical elements that interfere with the FunCSS tokeniser or parser. This technique has been suggested in a CSS specification draft [29]. However, due to the incompatibilities mentioned above, this solution could be inconvenient and prone to errors. As I want to create a convenient and safe language, I needed to find a different solution.

Nevertheless, in the current implementation of FunCSS, I used this technique for embedding JavaScript code into function declarations.

An ideal solution would enable the author to include any valid JavaScript code into a FunCSS file. The boundary between the FunCSS code and the JavaScript code need to be marked by some kind of delimiters. Delimiters should be easily identifiable, so that the author, the tokenizer, and even syntax highlighters can easily separate FunCSS code from JavaScript code. The CSS tokenizer already uses a mechanism for embedding URLs into CSS code: the `url(…)` functional notation is handled specially by the tokenizer [26]. A possible way of embedding JavaScript would be to use a similar approach: a `js(…)` functional notation could denote JavaScript code.

My supervisor, Ádám Siklósi suggested that literal JavaScript should be put between `{{…}}` blocks of double curly brackets. For example,

```
@def {
  sin($x:angle):number {{
    return Math.sin($x);
  }}
}
```

Opening double curly brackets are never used in CSS, so they do not interfere with CSS constructs. However, in literal JavaScript, double closing curly brackets are allowed. To know which `}}` pair closes the literal JavaScript block, the FunCSS compiler needs to identify the block structure of the embedded JavaScript code.

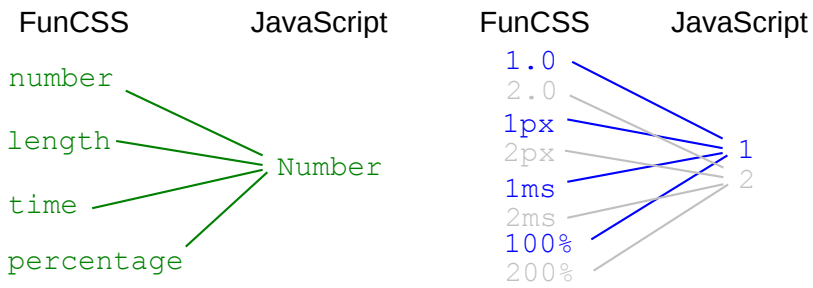
Identifying the block structure of JavaScript code is not trivial, as the code can contain opening and closing bracket characters within strings and regular expressions. Identifying regular expressions is also not trivial, because a `/` character can start either a regular expression or a division operator. To decide which token the `/` character starts, some information is needed about the context [12].

4.9.1 Conversion between FunCSS and JavaScript values

When embedded JavaScript code fragments work together with FunCSS constructs, they need to pass data between each other.

The type system of JavaScript and the type system of FunCSS are very different. FunCSS has three textual types: `<string>`, `<ident>` and `<url>`, while JavaScript has only one: `String`. Similarly, FunCSS has four numeric types: `<integer>`, `<number>`, `<percentage>` and `<dimension>`, while JavaScript has only

Figure 4.2
Mapping between FunCSS and JavaScript types and values



one: `Number`. FunCSS has the `<color>` type, which JavaScript does note have. On the other hand, JavaScript has `Boolean`, `Date` and `RegExp` types, which FunCSS does not have. Furthermore, both FunCSS and JavaScript have some structured types, but completely different ones: FunCSS has functional notations, blocks, combined and multiplied types, while JavaScript has arrays and objects.

One way to help passing data between FunCSS and JavaScript would be to extend the type system of one or both of the two languages. For example, I could add `Boolean`, `Date` and `RegExp` types to FunCSS. However, for the time being, I want to keep the set of basic types in FunCSS the same as those in CSS, in order to research how much can be achieved without adding any new basic types. Also, I could extend the type system of JavaScript with types from FunCSS (e.g. by adding lengths or urls), but I do not want to change the semantics of the embedded JavaScript code from the standard semantics of JavaScript. So I did not extend the type system of either FunCSS or JavaScript.

Instead of extending the type system of the two languages, I defined a mapping between FunCSS values and JavaScript values. In this mapping, I mapped values of several FunCSS types (the *source types*) to values of a single JavaScript type (the *target type*) (see Figure 4.2 for a few examples). I defined the mappings so that within a single FunCSS type, the mapping is a bijection between FunCSS values and JavaScript values. That is, the mapping from FunCSS values to JavaScript values is of the form

$$\text{FunCSS type} \times \text{FunCSS value} \to \text{JavaScript value},$$

and the mapping from JavaScript values to FunCSS values is of the form

$$\text{FunCSS type} \times \text{JavaScript value} \to \text{FunCSS value}.$$

My goal was to define the mappings so that they should be practical for the style sheet author. I judged what is practical based on my own programming experience. I tried to find the best mapping for each FunCSS type myself, however, later I might receive feedback about this from users of FunCSS. The mappings I defined can be seen in Table 4.3.

I mapped all numerical types of FunCSS (numbers, dimensions and percentages) to the `Number` type of JavaScript. Similarly, I mapped all textual types (idents, strings and URLs) of FunCSS to the `String` type of JavaScript.

For the mapping of dimensions, I needed some conversion rules that mirror the conversion rules between their units. For example, the `1in` and `2.54cm` lengths are equal, I needed to respect this in the mapping, and map them to the same `Number` value. For this reason, I have chosen a base unit for each quantity (`<length>`, `<angle>`, `<time>`, `<frequency>`, `<resolution>`). When converting a value, first it needs to be converted to the base unit, then its numeric value is taken and passed to the JavaScript code as a `Number`. The base units I chose can be found in Table 4.4.

For each quantity, I meant to choose a base unit which other JavaScript APIs already use. For example, many methods in the DOM use pixels as length units, so I chose pixel as the base unit. Similarly, as several mathematical functions in JavaScript use radians, I chose radian as the base unit for angles. For time values, I was considering seconds as the base unit because seconds probably would be intuitive for many people. However, as both the DOM and the JavaScript `Date` object uses milliseconds, I chose milliseconds instead. For frequencies, I have not found a suitable JavaScript API that I could consider in the decision. However, I chose `kHz` because it is the reciprocal of `ms`, the base unit of time. For resolutions, I used dots per pixel (`dppx`) as the base unit because there are some JavaScript APIs that use it, and because it is the reciprocal of `px`.

I also converted `<percentage>` values to numbers, so that `100%` is mapped to `1`. However, this solution is probably not the best. In CSS, the interpretation of a percentage depends on the context. For example, a percentage in the value of the `line-height:` property is a percentage of the font size, while a percentage in the argument of the `rgb(…)` functional notation is a percentage of the number `255`. To make this behavior easily achievable in custom FunCSS constructs, in the future, I want to introduce a way to define how percentages are interpreted. Then, the JavaScript mapping would use this interpretation when mapping percentage values.

All multipliers, except for Optional, are mapped to the `Array` type. I chose this because, based on my experience, operations with `Array` values are quite common in JavaScript development.

Table 4.3
Rules for mapping each FunCSS type to a JavaScript type

FunCSS type name	Example type	Example value	JavaScript type	Example value in JS
Simple types				
Keyword	`auto`	`auto`	String	`"auto"`
Ident	`<ident>`	`hello`	String	`"hello"`
String	`<string>`	`"Hello"`	String	`"Hello"`
Url	`<url>`	`url(funcss.org)`	String	`"funcss.org"`
Constant number	`400`	`400`	Number	`400`
Integer	`<integer>`	`10`	Number	`10`
Number	`<number>`	`3.14`	Number	`3.14`
Dimension	`<length>`	`3px` `3in`	Number	`3` `288`
Percentage	`<percentage>`	`75%`	Number	`0.75`
Color	`<color>`	`rgb(128,0,0)`	Object	`{r:128,g:0,b:0}`
Delimiter	`,`	`,`	String	`","`
Multipliers				
Optional	`<number>?`	`1` *nothing*	*depends*	`1` `undefined`
Zero or more	`<number>*`	`1 2` *nothing*	Array	`[1, 2]` `[]`
One or more	`<number>+`	`1 2`	Array	`[1, 2]`
Comma separated	`<number>#`	`1,2`	Array	`[1, 2]`
Combinators				
Juxtaposition	`<number> <length>`	`1.5 3px`	Array	`[1.5, 3]`
And	`<number>&&<length>`	`1.5 3px` `3px 1.5`	Array	`[1.5, 3]` `[1.5, 3]`

Inclusive or	`<number>||<length>`	`3px`
		`1.5 3px`
		`3px 1.5`
			Array	`[undefined, 3]`
				`[1.5, 3]`
				`[1.5, 3]`

Exclusive or	`<number>|<length>`	`1.5` `3px`	*depends*	`1.5` `3`

For the Optional multiplier I had two options in mind: either use an Array with zero or one element (as other multipliers use Arrays as well), or simply pass

Table 4.4 Base units of dimensions

Quantity	Base unit	Examples in JavaScript APIs
`<length>`	pixel (`px`)	`Event#pageX`
`<angle>`	radian (`rad`)	`Math.sin(…)`
`<time>`	millisecond (`ms`)	`window.setTimeout(…)`, `Date#getTime()`
`<frequency>`	kilohertz (`kHz`)	—
`<resolution>`	dots per pixel (`dppx`)	`window.devicePixelRatio`

the specified value and `undefined` if it is not given. Based on my experience, I think that the latter solution is more in accord with common JavaScript programming practices.

I chose to map all combinators to `Array`, except for Optional. For the first three combinators, the order of appearance of the elements is always in the order of declaration. For the Optional combinator, the type of the value is decided by the matching branch of the type.

Here is an example of how the mappings can be used. An author wants to define the `sin(…)`, which returns the sine of an angle. She wants to do the calculation from JavaScript, by calling the appropriate library function. She needs to specify the type of the argument and the return type of the function as follows.

```
1 sin(x:angle):number {{
2   return Math.sin(x);
3 }}
```

The type of the argument of `sin(…)` is `angle`, and the return type is `number`. The mapping defines how `angle` is converted to a JavaScript value: it is converted to radians, and passed as a `Number`. Then the JavaScript code does the calculation, and returns a `Number` value. Then the FunCSS code converts it to a FunCSS number.

As JavaScript is dynamically typed, the type of the values returned by the function cannot be known at compile time. For this reason, FunCSS must perform an implicit type conversion to convert the return value to the desired return type (in this case to `<number>`;). This conversion can theoretically be omitted if the compiler can decide through static analysis that the return

value cannot be anything but the desired type. However, this is generally not decidable.

4.10 Additional features

According to Question 6 of the survey, some other features should be considered for future inclusion into FunCSS. The support for defining new pseudoclasses, properties, at-rules and easings (timing functions for animations) received a generally positive feedback.

In Question 7, the best two items were nesting CSS rules and browser compatibility features. However, all other items received a positive feedback (extending CSS rules, mathematical formulas, mathematical functions, access to client-side data, automatic spriting, event handling).

4.10 Additional features

Section 5
Evaluation

5.1 Implementability

I have started to build an implementation of the FunCSS compiler. The source code is available on GitHub.[22] Though it is not complete, it already supports the definition of functions and the definition of variables (with a semantics that is still subject to change). It supports CSS qualified rules with simple selectors, and supports a few standard CSS properties.

I have also started to design and implement the standard library of FunCSS. It already contains some utility functions. For example, the two samples presented in 1.1 are actually working: they compile and run correctly.

The functional reactivity in the runtime environment is implemented using Meteor Tracker (see 2.2.1).

22. https://github.com/funcss-lang/funcss

5.2 Conformance to the goals

In 4.1, I presented the following design goals for FunCSS.

- a) It should be a declarative language, with rule-based and reactive paradigms
- b) It should have a human-friendly syntax.
- c) It should run fast enough for implementing animations
- d) It should contain abstractions for structuring code.
- e) It should be implementable in all widely used browsers.
- f) It should have full access to the Document Object Model.
- g) It should be Turing-complete.

In this section, I evaluate how I succeeded in realising these goals.

5.2.1 Declarative language with rule-based and reactive paradigm

FunCSS has become a declarative language. As JavaScript can be embedded into it, FunCSS is not a purely declarative language. However, if the author keeps the conventions of Tracker (see 2.2.1), the language will be purely declarative.

5.2.2 Human-friendly syntax

The survey that I made about the syntaxes confirms the syntaxes for definitions are fairly good.

5.2.3 Runtime performance

I conducted an experiment to measure the runtime performance of FunCSS for animations, compared to jQuery, d3.js and a pure JavaScript solution. The task was to animate 1000 red squares from 0 to 1000 pixels horizontally, with durations of 20, 21, ..., 1019 seconds. The codes used can be seen in Appendix C.

The FunCSS code, due to the currently incomplete implementation of the language, was different in two qualities from the other three. First, the styling

Table 5.1 Results of benchmark (frames per second)

Name	1000 squares with drawing	1000 squares without drawing	100 squares with drawing	100 squares without drawing
reference	59	58		59
pure JavaScript	21	48	48	59
D3.js	11	59	49..27	58
FunCSS	7	27	42	56
jQuery	8	16	35..27	56

of the elements were not set up algorithmically, but rather all 1000 rules for the 1000 squares were included in the source code. Second, with FunCSS, the animations were not configured to stop at 1000 pixels, but they continued indefinitely. This means that no checks were made whether the animations needs to stop.

I measured frames per second with the squares visible and with the squares hidden, to eliminate the cost of the redrawing. I also included a reference measurement with no animation. I made all measurements in Firefox 38. The results can be seen in Table 5.1.

The web browser limits the frame rate to 60 frames per second, and the reference measurement (when there was no animation) has yielded frame rates close to this. Animations created with FunCSS and jQuery performed similarly in all three sequences. However, d3.js performed better when the squares were drawn.

I suspect that d3js does some optimization when the squares are not visible, because it performed better than the pure JavaScript implementation.

It can be seen that FunCSS performed with 27 frames per second when 1000 calculations were executed per frame without drawing. This means that the FunCSS code could perform 27000 calculations per second (which involved calculating a simple linear functional relation).

Although a more extensive measurement would be required to give a detailed performance analysis, it can be seen from these few measurements that FunCSS can be used with animations that do not require more than a few hundred simultaneous calculations.

5.2.4 Abstractions for structuring code

Currently FunCSS only supports the definition of functions (and of variables, but the semantics of these have not been fixed). This single feature already provides a significant amount of abstraction possibilities. Functions can have either a JavaScript block or a FunCSS expression as their body, and overloading is possible both by input parameter type and output parameter type.

A good example of how overloading of functions can be used is the `linear(…)` function of the standard library. This function calculates a linear functional relationship. One of its definitions is the following:

```
1 linear(x:time, y0:length x0:time, y1:length x1:time):length {{
2   return y0 + (x-x0)*(y1-y0)/(x1-x0);
3 }}
```

It can be used to produce a linear motion. For example, the benchmark task in the previous section involved creating a box that is at 0 pixels at page load, and moves 1000 pixels within 20 seconds. This was achieved with `linear(…)`:

```
1 left: linear(time-since-load(), 0 0, 1000px 20s);
```

`linear(…)` can be used in various situations with various different input and output parameter types. However, currently for this, all combinations need to have a separate definition. It would be better if FunCSS supported generic programming, so that only one definition would suffice for several input-output type combinations.

5.2.5 Cross-browser compatibility

I have not tested FunCSS for cross-browser compatibility yet, as currently I am only building a prototype. However, I used the Tracker library, which has been tested for cross-browser compatibility by the developers of Meteor.

I am also considering to use jQuery in the runtime environment of FunCSS. jQuery has many cross-browser compatibility features, and it might not be useful to reimplement these from scratch in FunCSS.

5.2.6 Full access to the Document Object Model (DOM)

Theoretically, this condition is met, as JavaScript can be embedded into FunCSS, and JavaScript has full access to the DOM. However, it is rather complicated to create JavaScript functions which can cooperate with Tracker to ensure reactivity (see 2.2.1).

To make it easier for users of FunCSS to access the DOM in a reactive manner, a library should be created. According to my plans, this library will mimic the API of jQuery (and, it will originate from jQuery), but the getter methods will be reactive data sources. For example, the following call, which returns the `class` attribute of the element, would also set up a listener on the DOM, and rerun the current computation if the `class` attribute of the element is changed.

```
1 $(element).attr("class")
```

5.2.7 Turing-completeness

As JavaScript is a Turing-complete language, embedding JavaScript made FunCSS Turing-complete as well. However, just like in the previous section, it is not trivial to create an algorithm in a function and at the same time make the function a reactive data source (see 2.2.1). For everyday work, the standard library should provide utility functions to ease the work with reactive data sources.

5.3 Future work

FunCSS could be improved in the following directions:

Type system. The formalism of the type system should be developed, and checked with a proof assistant. The semantics of type inference, ambiguous types should be defined. A possible extension of the type system would be to define specificities of types, and in case of ambiguities, choose based on the specificities.

Percentages. Percentages are implemented as simple data types in the current version of FunCSS. However, a better solution would be to somehow add support for percentages to type system. In CSS, the context of a value defines how percentages are interpreted. FunCSS should also have support for defining the interpretation of percentages.

Custom selectors. A syntax should be developed for defining custom selectors. As I see it, the implementation of support for custom selectors would require a thorough analysis of the CSS cascading system, and a reimplementation of a significant part of it.

Preprocessor integration. It should be decided whether to integrate with a single preprocessor, or with several ones, and whether the integration should be mandatory. It is important to study how difficult it is to integrate one preprocessor and maintain the integration. Also one should study how important it is for users to easily migrate their existing code bases, which might have been written with various preprocessors.

Optimizations. Optimize the compiler for runtime performance primarily, and for compilation speed also

Event handling. Support for event handling could be added.

Conclusion

I designed an extension of CSS based on the idea to enable the definition of custom functions. I used feedback from a survey among web developers to find out the optimal syntax for function definitions and variables. I sought to design a language which is declarative, follows the rule-based and reactive paradigms of CSS, has a human-friendly syntax, can be implemented efficiently in all widely used web browsers, and can be considered a full-featured programming language.

The resulted language supports the functional reactive paradigm, and has a type system based on regular grammars. Literal JavaScript code fragments can be embedded into the language as bodies of functions.

I made a compiler and a runtime for a subset of this new designed language. The language and the implementation performed well in several of the above mentioned goals, however, there is still much work to be done.

S.D.G.

Appendices

A A formalism of the CSS type system

I describe here a formal type system that I have developed for FunCSS. It is based on the Value Definition Syntax of the CSS specification [25]. I will refer to this type system as *FunCSSts* (an abbreviation for *FunCSS type system)*.

A.1 Notations

I will use the common notations for type systems, described for example in [13]. I used values with I used Γ to denote environments, *M,N* to denote terms, *A,B* to denote types, and $\Gamma \vdash \ldots$ to denote that a judgement follows from the environment. The following judgements are used:

Table 5.1 Judgements in FunCSSts

Judgement	Meaning
$\Gamma \vdash \diamond$	The environment is well-formed
$\Gamma \vdash A$	Type A is well-formed
$\Gamma \vdash M : A$	Term M inhabits type A

I use the following notations in formal grammars. I use words with *this font* to denote nonterminals, and words with `this font` to denote terminals. The ::= relation denotes grammar rules, the ⇒ relation means that the right hand side is derivable from the left hand side in one step, and the ⇒* relation is the transitive closure of ⇒ [14].

A.2 Value grammar

In Table 5.1, I present the FunCSSts Value Grammar (FunCSStsVG), the grammar of terms in the FunCSSts type system. It generates all lists of component values that can be used in property values.

Table 5.2 The FunCSSts Value Grammar (FunCSStsVG)

componentValueList ::= componentValue | componentValue componentValueList
componentValue ::= basic | structured
basic ::= ident | string | url | number | dimension | percentage | hash | delimiter
structured ::= block | functionalNotation
ident ::= a | b | ... | aa | ab | ...
string ::= "a" | "b" | ... | "aa" | "ab" | ...
url ::= url(a) | url(b) | ... | url(aa) | url(ab) | ...
number ::= integer | 0.0 | 0.1 | ... | 0.01 | ... | -0.01 | ...
integer ::= 0_i | 1_i | ... | -1_i | -2_i | ...
dimension ::= 0a | 0b | ... | 1a | ... | -1a | ... | 0.0a | ... | 0aa | ...
percentage ::= 0% | 1% | ... | 0.0% | 0.1% | ... | -0.1% | ...
hash ::= #a | #b | ... | #aa | #ab | ...
delimiter ::= , | /
block ::= (componentValueList) | [componentValueList] | { componentValueList }
functionalNotation ::= functionToken componentValueList)
functionToken ::= a(| b(| ... | aa(| ab(| ...

I make some notes about this grammar.

1. The starting nonterminal is *componentValueList*.
2. This grammar does not generate empty component value lists. Probably in a later version these should be added.

3. The only production that can yield several terminals is *componentValueList*. This is in line with the fact that in FunCSSts, all expressions are component value lists, and the only operation is list concatenation.
4. I use the word *ident* to refer to any valid identifier, and the word *keyword* to refer to an identifier with a special meaning.
5. I do not define here explicitly the set of keywords, strings, urls, numbers, dimensions, percentages, hashes and function tokens, but I assume that the tokenizer will output these values correctly. The definition of these sets follow from the definition of the tokenizer, which is available in the specification [26, s4].
6. The tokenizer outputs numbers and dimensions with a type flag, which can be either *integer* or *number* depending on the number format [26, s4.3.12]. I denote the *integer* type flag with an $_i$ in the subscript for numbers. For dimensions, I do not use the type flag, so I did not include it in the grammar.
7. The *componentValueList* production of FunCSStsVG generates the same token lists as the `value` production of the css core grammar [19, s4.1.1], with a few differences:
   - FunCSStsVG does not generate terms with at-keywords.
   - FunCSStsVG does not generate terms with unicode ranges.
   - FunCSStsVG does not generate terms whith delimiters or delimiter-like tokens besides commas and slashes.
   - FunCSStsVG does not generate terms with whitespaces.

   The reason for these differences is that property value grammars never allow at-keywords, unicode ranges and delimiters other than commas and slashes; and they always ignore whitespaces.
8. *hash* is a textual type, which (in property values) is solely used to represent colors. The tokenizer (and the core parser as well) returns all kinds of hashes, regardless of whether they represent valid colors. Accordingly, I include all hashes in the syntax, and restrict them to valid colors in the type rules.

A.3 Type grammar

Table 5.3 The FunCSSts Type Grammar (FunCSStsTG)

type	::=	*simpleType* | *functionalNotationType* | *combinedType* | *multipliedType*
simpleType	::=	*keywordType* | *basicType* | *typeReference* | *constantIntegerType* | *delimiterType*
keywordType	::=	*ident*[1]
basicType	::=	`<ident>` | `<string>` | `<url>` | `<integer>` | `<number>` | `<length>` | `<angle>`

	`<time>` | `<frequency>` | `<resolution>` | `<percentage>` | `<color>`
typeReference ::=	`<generic-family>` | ... | `<'border-color'>` | ...
constantIntegerType ::=	`0`$_i$ | `1`$_i$ | ... | `-1`$_i$ | `-2`$_i$ | ...
delimiterType ::=	*delimiter*[1]
functionalNotationType ::=	*functionToken*[1] *type* `)`
combinedType ::=	*juxtaposition* | *inclusiveOr* | *and* | *exclusiveOr*
juxtaposition ::=	`[` *type type* `]`
inclusiveOr ::=	`[` *type* `||` *type* `]`
and ::=	`[` *type* `&&` *type* `]`
exclusiveOr ::=	`[` *type* `|` *type* `]`
multipliedType ::=	*optional* | *oneOrMore* | *zeroOrMore* | *range* | *commaSeparated*
optional ::=	*type* `?`
oneOrMore ::=	*type* `+`
zeroOrMore ::=	*type* `*`
range ::=	*type* `{1,1}` | *type* `{1,2}` | ... | *type* `{2,2}` | ...
commaSeparated ::=	*type* `#`

[1] See Table 5.1 for the definition of these productions

Some notes about the FunCSSts Type Grammar (FunCSStsTG):

1. In some productions (*keywordType*, *delimiterType* *functionalNotationType*) I refered to productions defined in FunCSStsVG. This is a purely syntactical abbreviation which follows from the fact that the CSS VDS is homoiconic [15].
2. Values of *typeReference* come from two sources: types defined for convenience, and value types of properties. The latter is denoted with apostrophes inside the angle brackets. The exact contents of these sets is outside the current scope of the type system.
3. This grammar does not implement `[...]` groups. The reason why I left them out is that they are purely syntactical in the original CSS Value Definition Syntax.

4. Multipliers are defined in FunCSStsTG as binary operators for simplicity, and I implemented them in FuncSS this way. In the original VDS, however, they are *n*-ary.
5. Though `[...]` groups of the VDS are not used, I used the same characters to denote the binary operators of multiplied types. This way FunCSStsVG is a subset of the VDS syntax.

A.4 Inference rules

Here I define the inference rules for the FunCSSts type system.

Table 5.4 The FunCSSts Inference Rules (FunCSStsIR)

Environment

(Env ø) (5.1)

$$\frac{}{\text{ø} \vdash \diamond}$$

Keywords

(Type keyword) (5.2)

$$\frac{\Gamma \vdash \diamond \quad \mathit{ident} \Rightarrow^* A}{\Gamma \vdash A}$$

(Val keyword) (5.3)

$$\frac{\Gamma \vdash \diamond \quad \mathit{ident} \Rightarrow^* A}{\Gamma \vdash A : A}$$

Simple values (ident, string, url, integer, number, percentage)

(Type ident)[1] (5.4)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{<ident>}}$$

(Val ident)[1] (5.5)

$$\frac{\Gamma \vdash \diamond \quad \mathit{ident} \Rightarrow^* M}{\Gamma \vdash M : \texttt{<ident>}}$$

Dimensions (length, angle, time, frequency, resolution)

(5.6) (5.7) (5.8)

(Type length)[2]

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{<length>}}$$

(Val length)[2]

$$\frac{\Gamma \vdash \diamond \quad \mathit{dimension} \Rightarrow^* M \quad \text{unit}(M) \in \mathit{LengthUnits}}{\Gamma \vdash M : \texttt{<length>}}$$

(Val length0)[2]

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{0} : \texttt{<length>}}$$

Color

(Type color) (5.9)

$$\frac{\Gamma \vdash \diamond}{\Gamma \vdash \texttt{<color>}}$$

(Val colorHex) (5.10)

$$\frac{\Gamma \vdash \diamond \quad \mathit{hash} \Rightarrow^* M \quad \text{value}(M) \in \mathit{Hex}^3 \cup \mathit{Hex}^6}{\Gamma \vdash M : \texttt{<color>}}$$

(Val colorNamed) (5.11)

$$\frac{\Gamma \vdash \diamond \quad M \in \mathit{ColorNames}}{\Gamma \vdash M : \texttt{<color>}}$$

(Val colorRGB)[3] (5.12)

$$\frac{\Gamma \vdash M : \texttt{rgb([<integer>[,[<integer>[,<integer>]]]])}}{\Gamma \vdash M : \texttt{<color>}}$$

Functional notation

(Type fN) (5.13)

$$\frac{\Gamma \vdash A \quad \mathit{functionToken} \Rightarrow F}{\Gamma \vdash FA)}$$

(Val fN) (5.14)

$$\frac{\Gamma \vdash M : A \quad \mathit{functionToken} \Rightarrow F}{\Gamma \vdash FM) \ : FA)}$$

Combinators

(Type juxtaposition) (5.15)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash [AB]}$$

(Val juxtaposition) (5.16)

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : [AB]}$$

(Type exclusiveOr) (5.17)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash [A \mid B]}$$

(Val exclusiveOr1) (5.18)

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : [A \mid B]}$$

(Type exclusiveOr2) (5.19)

$$\frac{\Gamma \vdash N : B}{\Gamma \vdash N : [A \mid B]}$$

(Type and) (5.20)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash [A \,\&\&\, B]}$$

(Val and12) (5.21)

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : [A \,\&\&\, B]}$$

(Val and21) (5.22)

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash NM : [A \,\&\&\, B]}$$

(Type inclusiveOr) (5.23)

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash [A \,||\, B]}$$

(Val inclusiveOr1) (5.24)

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : [A \,||\, B]}$$

(Val inclusiveOr2) (5.25)

$$\frac{\Gamma \vdash N : B}{\Gamma \vdash N : [A \,||\, B]}$$

(Val inclusiveOr12) (5.26)

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : [A \,||\, B]}$$

(Val inclusiveOr21) (5.27)

$$\frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash NM : [A \,||\, B]}$$

[1] Same for string, url, integer, number, percentage.

[2] Same for angle, time, frequency, resolution

[3] The expression used here is the VDS expression `rgb(<integer>,<integer>,<integer>)`, written using the binary juxtaposition combinator. Similar rules apply for `rgba(…)`, `hsl(…)`, `hsla(…)`.

Some notes on the inference rules:

1. As this type system does not include variables yet, the environment is essentially not used.
2. I did not formulate the inference rules for multipliers. Including multipliers in the type system would have required the inclusion of empty terms and empty types. That would have required further analysis, which I leave for further research.

B Survey results about FunCSS features

I conducted a survey between 28 Apr and 26 May 2015 about the possible features of FunCSS. The link to the survey was posted on 28 Apr to two Meetup Groups, BudapestJS and Budapest Frontend Meetup. I received 32 submissions.

1. Which one is a good syntax for variables?

```
$x
```

```
@x
```

```
%x
```

```
x
```

Other suggestions:

```
x =
```

2. Which one is a good syntax for defining variables? *(A variable x with type "number" and value 14.)*

```
@def $x 14;
```

```
@def $x:<number> 14;
```

```
@number $x 14;
```

```
$x: 14;
```

Other suggestions:

```
x = 14
x = 14
x = 14
x=14
```

3. Which one is a good syntax for defining functions? *(A function f which returns the sine of its argument.)*

```
@function f($x) sin($x);
```

```
@fun f($x) sin($x);
```

```
@fun f($x) = sin($x);
```

```
f($x) = sin($x);
```

Other suggestions:

```
(x) => sin x
@function f($x) = sin($x);
```

4. Which one is a good syntax for specifying the type of the argument of a function?

```
@fun f(x:number) = sin($x);
```

```
@fun f(x:<number>) = sin($x);
```

```
@fun f($x:<number>) = sin($x);
```

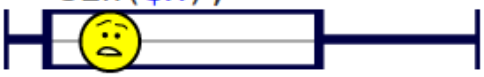
```
@fun f(<number> $x) = sin($x);
```

Other suggestions:

```
@fun f($x:Number>) sin($x);
dynamic types
f($x:number) = sin($x);
why do you want types?
```

5. Which one is a good syntax for specifying the return type of a function?

```
@fun f($x):<number> = sin($x);
```



```
@fun f($x):number = sin($x);
```

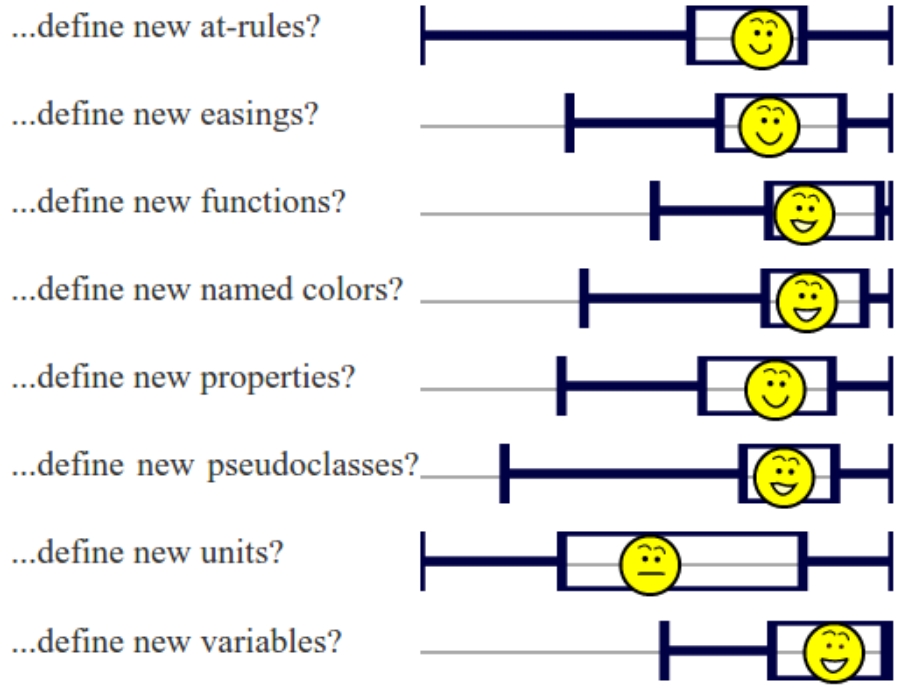
```
@fun <number> f($x) = sin($x);
```

```
@number f($x) = sin($x);
```

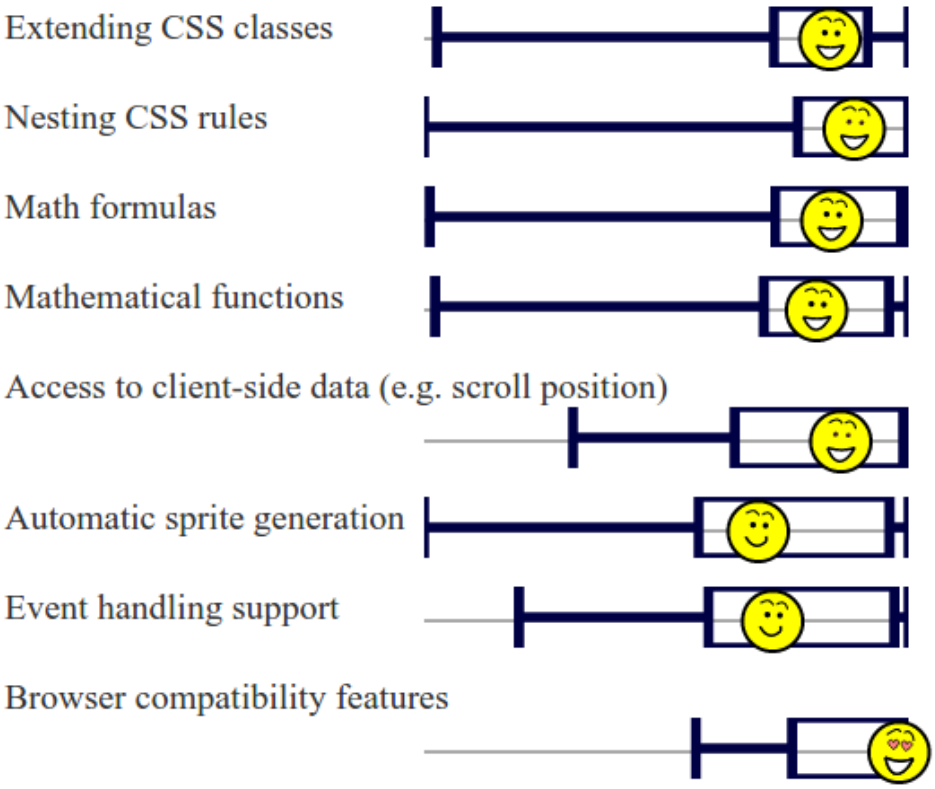
Other suggestions:

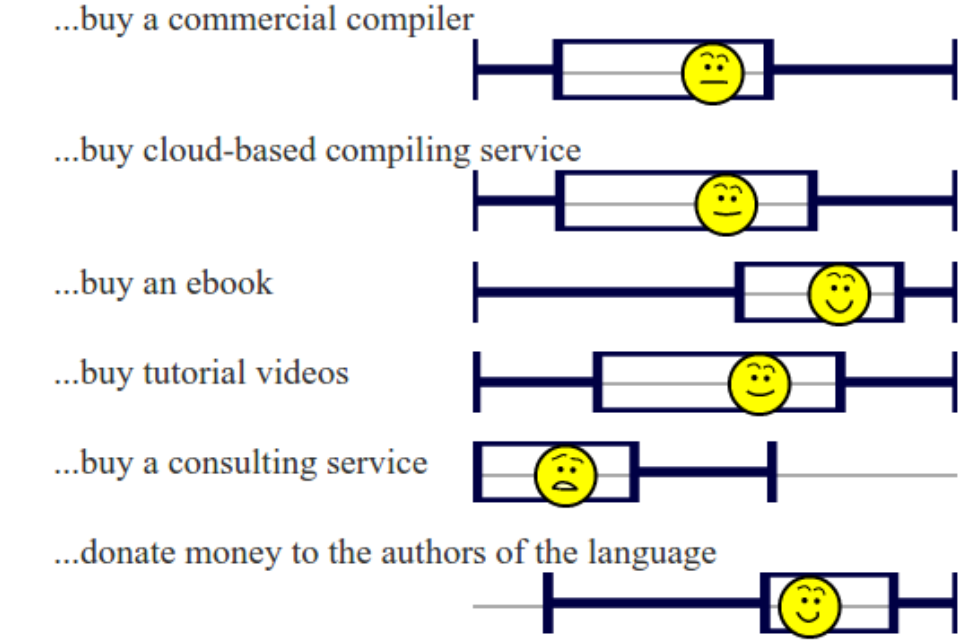
```
dynamic types
```

6. Would you like if, in an extended version of CSS, you could... *(Please skip those you're not familiar with.)*



7. Which other features could be useful in an extended CSS language? *(Please skip those you're not familiar with.)*





C Codes used for benchmarking

C.1 Pure JavaScript

CSS

```
.sq {
    width: 10px;
  height: 10px;
  background: red;
  position: absolute;
  display: none;
  //display: inline-block;
}
```

JavaScript

```
var N = 1000;
var c = 20000;
var deltaC = 1000;
var lengthInPx = 1000;
var ds = []
for (var i = 0; i < N; ++i) {
    var d = document.createElement("div");
    d.setAttribute("class", "sq");
    d.style.top = i*20 + "px";
    d.data_speed = lengthInPx / (c+i*deltaC);
    ds.push(d);
    document.body.appendChild(d);
}
var start = +new Date();
var stoppedCount = 0;
var stopped = false;
function tick() {
    var deltaT = (new Date() - start);
    var shouldBeStopped = Math.floor((deltaT - c) / deltaC) + 1;
    if (shouldBeStopped > stoppedCount) {
        if (shouldBeStopped >= N) {
            stopped = true;
            shouldBeStopped = N;
        }
        for (var i = stoppedCount; i<shouldBeStopped; ++i) {
            ds[i].style.left = lengthInPx + "px";
        }
        stoppedCount = shouldBeStopped;
    }
    for (var i = stoppedCount; i < N; ++i) {
        var d = ds[i];
        d.style.left = deltaT * d.data_speed + "px";
    }
    if (!stopped) {
        setTimeout(tick, 10);
    }
}
tick();
```

C.2 FunCSS

CSS

```
.sq {
    width: 10px;
    height: 10px;
    background: red;
    position: absolute;
    display: none;
    //display: inline-block;
}
```

JavaScript

```
var N = 1000;
var c = 20000;
var deltaC = 1000;
var lengthInPx = 1000;
var ds = []
for (var i = 0; i < N; ++i) {
    var d = document.createElement("div");
    d.setAttribute("class", "sq " + "sq-" + i);
    //ds.push(d);
    document.body.appendChild(d);
}
```

FunCSS

```
@import "vanilla-1.0";

.sq-0 {
    top: 0px;
    left: linear(time-since-load(), 0 0, 1000px 20s);
}
.sq-1 {
    top : 20px;
    left: linear(time-since-load(), 0 0, 1000px 21s);
}
.sq-2 {
    top : 40px;
    left: linear(time-since-load(), 0 0, 1000px 22s);
}
…
.sq-999 {
    top : 19980px;
    left: linear(time-since-load(), 0 0, 1000px 1019s);
}
```

C.3 d3.js

CSS

```
.sq {
    width: 10px;
    height: 10px;
    background: red;
    position: absolute;
    display: none;
    //display: inline-block;
}
```

JavaScript

```
var N = 1000;
var c = 20000;
d3.select("body")
.selectAll(".sq")
    .data(d3.range(0, N))
    .enter().append("div")
        .classed("sq", true)
        .style("top", function(d){return d*20+"px"})
    .transition()
        .ease("linear")
        .duration(function(d){return c+d*1000})
        .style("left", "1000px");
```

C.4 jQuery

CSS

```
.sq {
    width: 10px;
    height: 10px;
    background: red;
    position: absolute;
    //display: none;
    display: inline-block;
}
```

JavaScript

```
var N = 1000;
var c = 20000;
for(var i=0; i<N ; ++i ) {
    $("body").append($( "<div class='sq' style='top:"+i*20+"px'></di
        {left: 1000}, {duration: (c+=1000), easing: "linear"}
    ));
}
```

References

[1] H. Lie and B. Bos, 'The CSS Saga', in *Cascading style sheets*. Harlow, England: Addison-Wesley, 1999. [Online]. Available: http://www.w3.org/Style/LieBos2e/history/Overview.html. [Accessed: 31- May- 2015]

[2] N. Sullivan, 'CSS doesn't suck, you're just doing it wrong.' 2009 [Online], Available: https://web.archive.org/web/20120105135752/http://www.stubbornella.org/content/2009/02/12/css-doesn%E2%80%99t-suck-you%E2%80%99re-just-doing-it-wrong/ [Accessed: 2012- 01- 05]

[3] E. Bainomugisha, A. Carreton, T. Cutsem, S. Mostinckx and W. Meuter, 'A survey on reactive programming', *CSUR*, vol. 45, no. 4, pp. 1-34, 2013.

[4] 'List of suggested extensions to CSS', W3C Note, 1998, sec. 28. [Online] Available: http://www.w3.org/TR/NOTE-CSS-potential#id05684046681 [Accessed: 15- Jun- 2015]

[5] B. Bos, 'Why "variables" in CSS are harmful?', 2008, [Online] Available: http://www.w3.org/People/Bos/CSS-variables. [Accessed: 15-Jun-2015]

[6] C. Coyier, 'Poll Results: Popularity of CSS Preprocessors', 2012 [Online] Available: https://css-tricks.com/poll-results-popularity-of-css-preprocessors/. [Accessed: 15-Jun-2015]

[7] C. Chedeau, 'React: CSS in JS', 2014 [Online] Available: https://speakerdeck.com/vjeux/react-css-in-js. [Accessed: 15-Jun-2015]

[8] H. Lie, 'Cascading Style Sheets', PhD, University of Oslo, 2005.

[9] CSS Working Group Wiki, 'Functional Notation Systematization [CSS Working Group Wiki]'. [Online]. Available: https://wiki.csswg.org/ideas/functional-notation. [Accessed: 04- Jun- 2015]

[10] http://www.glazman.org/weblog/dotclear/index.php?post/2012/08/17/CSS-Variables%2C-why-we-drop-the-%24foo-notation

[11] http://www.xanthir.com/blog/b4KT0

[12] T. Disney, N. Faubion, D. Herman and C. Flanagan, 'Sweeten your JavaScript', *Proceedings of the 10th ACM Symposium on Dynamic languages - DLS '14*, 2014.

[13] L. Cardelli, 'Type Systems', in *Handbook of Computer Science and Engineering*, Chapter 103. CRC Press, 1997, [Online]. Available: http://www.cs.colorado.edu/~bec/courses/csci5535/reading/cardelli-typesystems.pdf. [Accessed: 1- Jun 2015]

[14] T. Jiang et al., 'Formal grammars and languages', [Online]. Available: http://www.cs.ucr.edu/~jiang/cs215/tao-new.pdf. [Accessed: 2- Jun- 2015]
[15] C2 Wiki, 'Definition of homoiconic', [Online]. Available: http://c2.com/cgi/wiki?DefinitionOfHomoiconic. [Accessed: 1- Jun- 2015]
[16] A. Deveria, 'calc()', *Can I use...* [Online], Available: http://caniuse.com/#feat=calc. [Accessed: 15- Jun- 2015]

CSS standards used

[17] 'Cascading Style Sheets, level 1', [First public version]. W3C Recommendation, 1996 [Online]. Available: http://www.w3.org/TR/REC-CSS1-961217 [Accessed: 31- May- 2015]
[18] 'Cascading Style Sheets, level 1', W3C Recommendation, 1996 [Online]. Available: http://www.w3.org/TR/REC-CSS1 [Accessed: 31- May- 2015]
[19] 'Cascading Style Sheets Level 2 Revision 1 (CSS 2.1) Specification', W3C Recommendation, 2011 [Online]. Available: http://www.w3.org/TR/CSS2 [Accessed: 30- May- 2015]
[20] 'Selectors Level 3', W3C Recommendation, 2011 [Online]. Available: http://www.w3.org/TR/css3-selectors/. [Accessed: 30- May- 2015]
[21] 'CSS Image Values and Replaced Content Module Level 3', W3C Working Draft, 2011 [Online]. Available: http://www.w3.org/TR/2011/WD-css3-images-20110712/ [Accessed: 30- May- 2015]
[22] 'Media Queries', W3C Recommendation, 2012 [Online] Available: http://www.w3.org/TR/css3-mediaqueries/. [Accessed: 15- Jun- 2015]
[23] 'CSS Object Model', W3C Working Draft, 2013 [Online]. Available: http://www.w3.org/TR/cssom/. [Accessed: 15-Jun-2015]
[24] 'CSS Transforms Module Level 1', W3C Working Draft, 2013 http://www.w3.org/TR/css-transforms-1/
[25] 'CSS Values and Units Module Level 3', W3C Candidate Recommendation, 2013 [Online]. Available: http://www.w3.org/TR/css3-values/. [Accessed: 30- May- 2015]
[26] 'CSS Syntax Module Level 3', W3C Candidate Recommendation, 2014 [Online]. Available: http://www.w3.org/TR/css-syntax-3-20140220/. [Accessed: 1- Jun- 2015]
[27] 'CSS Cascading and Inheritance Level 3', W3C Candidate Recommendation, 2015 [Online]. Available: http://www.w3.org/TR/css-cascade-3-20150416/. [Accessed: 1- Jun- 2015]
[28] 'CSS Custom Properties for Cascading Variables Module Level 1', W3C Editor's draft, 2015 [Online]. Available: http://www.w3.org/TR/css-variables-1/. [Accessed: 4- Jun- 2015]

[29] 'CSS Extensions', W3C Editor's draft, 2015 [Online]. Available: http://dev.w3.org/csswg/css-extensions/ [Accessed: 4- Jun- 2015]